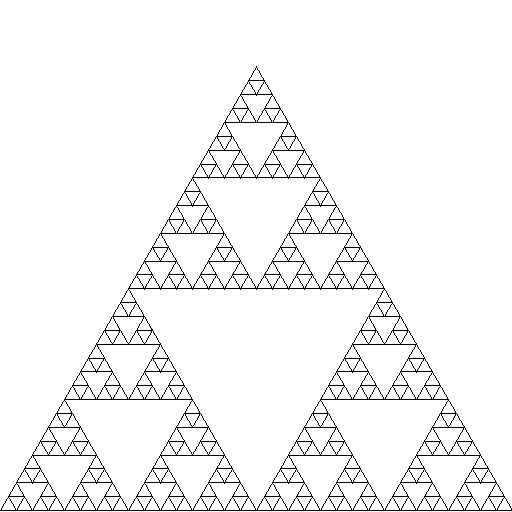
**Numbrix Puzzle**

Recursion is, simply put, the process of a method calling itself. A recursive method calls itselfuntil a certain **base case** has been reached, at which point the method returns an actual value (rather than another recursive method call).

Recursion makes some problems significantly simpler (simpler than the iterative solutions, that is). Recursion allows us to break up a complex problem into smaller and smaller parts, the simplest of which we should be able to solve immediately. Recursion is an entirely new way of thinking and solving problems, and it will take quite a bit of practice to get good at it.

A recursive algorithm must have a base case, otherwise it will recursively call itself forever. When a method is called, it is placed on the **stack\****.* If a method is called recursively without ever reaching the base case, the computer will eventually run out of stack memory, and you get a ***StackOverflow****Exception*. Sound familiar?

*\*The* ***stack*** *(or call stack) is a special region of your computer's memory that stores information on the active methods in your program. It is used to keep track of where a method should return control when it's done running.*

**Thinking recursively**

1. Describe with recursion how you would read an entire book.
2. Describe with recursion how you would eat a sandwich.
3. Describe with recursion how you would build a pyramid.

*Recursive artwork*

**Writing recursive methods**

1. Complete the method public int numEars(int bunnies) to calculate the number of ears on bunnies number of bunnies (without loops or multiplication).

//what is the base case – the simplest possible solution that you can solve now?

1. Complete the method public void countdown(int num) that will print a countdown, as shown below (without using a loop):

countdown(5) should print: 5, 4, 3, 2, 1, blastoff!

1. Write a recursive method to calculate the factorial of the integer parameter. Don't calculate the factorial of a large number – factorials get big fast.
2. Complete the recursive method public String cheerlead(String str, int i) that returns a String with str repeated inumber of times.

cheerlead("Go team!", 3) >>> "Go team! Go team! Go team!"

1. Write a method that computes (recursively) the value of one number to the power of another number. **Hint:** if exponent is 0, return 1 - otherwise, break up the problem into smaller pieces.
2. (Riddle) The police are looking for a suspect in a bank robbery. All they know is that he's staying in a certain hotel room. They burst into the room, and inside are four people playing Monopoly: a lawyer, a fireman, a banker and a doctor. They immediately arrest the fireman. Why?
3. Write a recursive method to compute the nthnumber in the Fibonacci sequence. The first two values in the sequence are 0 and 1 (F0 = 0, F1 = 1). Each subsequent value is the sum of the previous two values (Fn = Fn-2 + Fn-1), so the whole sequence is: 0, 1, 1, 2, 3, 5, 8, 13, 21 and so on.
4. Write a recursive method that will print the following pattern:

printPattern(16) should print 16, 11, 6, 1, -4

printPattern(10) should print 10, 5, 0, -5

1. Compute recursively (without loops) the number of 'a' characterss in the String parameter.

countNumA("aaHELLOa") >>> 3

1. Write a method public void printAtoBbyC(int a, int b, int c) that prints all the numbers from a to b, in increments of c (separated by a space). You can assume this will always be possible (that the numbers supplied by the user conform to expectations).

printAtoBbyC(10, 30, 5) should print: 10 15 20 25 30

1. Write a recursive method that returns how many digits of an integer parameter are odd. **Note:** you shouldn't need Strings or String methods for this, just arithmetic operations.

countOdds(123456) >>> 3

1. Given a non-negative integer, compute (recursively) the sum of its digits.

sumDigits(128) >>> 11 //1 + 2 + 8 == 11

1. Complete the "Recursion tracing #1" worksheet, in the lab folder.

**Numbrix Puzzle Solver**

***Note:*** *this lab is essentially a "proof of concept". This lab presents a* **very** *challenging task, from a computational standpoint, and demonstrates an extremely fast and elegant solution that utilizes recursion. The overwhelming majority of AP students will not understand the recursion used at this point, and that is okay. Recursion is strange at first, and this lab attempts to demonstrate that it IS a useful problem-solving technique. Follow the instructions carefully and do your best to understand what is going on, if you get stuck I will show you a solution.*

In this lab you will write a program that solves Numbrix puzzles. Numbrix puzzles were invented by Marilyn vos Savan, the author of the "Ask Marilyn" column in Parade Magazine. Numbrix puzzles can be found at http://www.parade.com/numbrix and various other online sites.

Numbrix puzzles consist of a grid with numbers in some of the cells. The puzzle supplied for this assignment is depicted on the right.

A solved Numbrix puzzle contains all the numbers from 1 to *rows* x *cols* (in this example 9 x 9 = 81) filled in. The original numbers must be unchanged, and consecutive numbers must be next to each other either vertically or horizontally.



The solution to this particular puzzle is depicted on the left. The most straightforward way to attack this problem is to try all the possible positions for the remaining numbers and see which ones solve the puzzle, in an iterative (i.e. for loops) fashion. This can also be accomplished with a recursive "depth-first" search.

However, in this puzzle, there are 81 - 16 = 65 numbers to place. Therefore, there are 65! (65 factorial) possible placements for the remaining 65 numbers, only one of which is correct. 65! is a pretty large number. It's even greater than 1090, a one followed by 90 zeros.

How long would it take to try 1090 different possible solutions? Let's assume that we have a smoking fast program running on a really fast computer that can compute a quadrillion (1,000,000,000,000,000 = 1015) possible number placements every second. Then it would take more than 1090 / 1015 = 1075 seconds for our program to complete. But how long is 1075 seconds?

1075/ 60 / 60 / 24 / 365 is more than 1067 years. Since the AP Exam is in May, we need a faster algorithm.  
  
The code you are going to write solves a Numbrix using a recursive depth-first search **with pruning**. "Pruning" means that your code will check each number before it is placed into the puzzle and if the number doesn't "fit", it will not explore that branch of the search any further. Using this strategy, a puzzle can be solved in seconds!

Our search for a solution begins with the solve method which iterates through each element (row r and column c) of grid and attempts to solve the puzzle by starting with a 1 in that location. It does this by calling recursiveSolve(r, c, 1) to attempt a solution beginning with a 1 at row r and column c.

recursiveSolve is the method that performs the recursive depth-first search. After placing a number n, it recursively attempts to place the number n+1 in the location above, below, left, and right of the location where it placed n. It continues to attempt placing subsequent numbers until the last number is successfully placed.

Obviously, the program is unlikely to be successful on every placement attempt. When a number n can't be successfully placed at row r and column c, the recursiveSolve method returns and lets the method invocation that called it attempt a different placement. recursiveSolve returns when:

* either r or c is outside of grid. This is base case one.
* the current location contains 0 (empty), but the number to be placed was used elsewhere in the original puzzle. This is base case two and the first "pruning" situation.
* the current location contains a number, but it's not equal to the number to be placed. This is base case three and the second "pruning" situation.
* the puzzle is solved. In this case we print the solution and return to look for more solutions. This is base case four.
* the four recursion calls have been completed.

1. You will be using two classes in this project:  
   1. NumbrixMain – the application (runner) class for this project. This class has been completely written for you. **Do not make any additions or changes to it.**
   2. Numbrix – objects of this class represent a Numbrix puzzle.  **This is where you will place your code.**
2. The Numbrix class already contains two instance variables: grid and used. The grid variable will contain the puzzle, and used will indicate if a number was used in the original puzzle. These are the only instance variables needed. **Do not add any additional instance variables.**
3. First, complete the constructor for theNumbrixclass. It should do the following:  
   1. Create a Scanner object to input data from the file named in filename. Use new Scanner(new File(fileName)) to instantiate your Scanner.
   2. Read in the number of rows and columns in the puzzle. These are the first and second numbers in the data file, respectively.
   3. Instantiate the grid 2D array to have the input number of rows and columns.
   4. Instantiate the used array to have (rows \* columns + 1) elements. The extra element allows you to use the numbers in the puzzle as indices. Otherwise you would need to subtract 1 to prevent an ArrayOutOfBoundsException. The element at index 0 is ignored.
   5. Input the puzzle numbers and use them to fill in grid and used. used[num] should be true if num is in the original input puzzle. It should be false otherwise.
4. Complete the toString method to return a String with the grid data. The requirements for the string are as follows:  
   1. The data must be in row-major (ordinary) order.
   2. 0s in the array should be represented with dash (minus sign) characters.
   3. There must be one tab character after each number or dash.
   4. There must be one new line character after each row.
   5. There must be no extraneous spaces of other characters.
5. Test your program and correct any errors. It should produce the following output (tab width may vary):  
     
   49 - 51 - 63 - 69 - 71  
   - - - - - - - - -  
   47 - - - - - - - 77  
   - - - - - - - - -  
   45 - - - - - - - 81  
   - - - - - - - - -  
   43 - - - - - - - 19  
   - - - - - - - - -  
   41 - 37 - 9 - 13 - 15
6. Complete the solve method. This method should attempt to solve the puzzle by starting with a 1 in each grid element. For each element of grid, solve should call recursiveSolve(r, c, 1)† where r and c are the current row, column of the grid location in which you'll attempt to place the 1.
7. Test your new solve method by adding temporary code to recursiveSolve that prints r, c, and n. Make sure that solve calls recursiveSolve for all possible r and c. Also ensure that n is always 1.  
     
   When you are satisfied that solve is working correctly, remove the temporary code from recursiveSolve.
8. Complete the recursiveSolve method. This is the recursive method that performs the depth-first search for solutions. **Follow the instructions carefully to complete this method.** Complete it as follows:  
   1. Make sure that r and c specify an element inside the grid. If r is an invalid row index, or c is an invalid column index, then return. This is base case one.
   2. Create and initialize a boolean variable named zero. This variable needs to be true iff grid[r][c] contains a 0. **Do this with one statement of the following form:** boolean zero = ... ;
   3. The next two base cases (d) and (e) consist of situations where it does not make sense to place n in grid[r][c]. They constitute the pruning discussed earlier.
   4. If zero is true, but n is in the original puzzle, then n can't be placed in grid[r][c] because it is already used elsewhere. In this case, your method should return. Utilize used to determine if n is in the original puzzle. This is base case two.
   5. If zero is false and grid[r][c] contains a number other than n, then your method should return. This is base case three.
   6. At this point, it makes sense to store n in grid[r][c]. Go ahead and do that.
   7. Now check to see if the puzzle is solved. See if n equals the product of the number of rows and columns in grid. If so, you should print this with System.out.println(this); which in turn calls toString implicitly. This is the fourth base case. Don't return yet though because we need to remove the number at grid[r][c] and look for other solutions.
   8. If the puzzle isn't solved then make four recursive calls. There four calls should specify the element that is up, down, left, and right of the current element. The 3rd parameter should be n + 1.
   9. Finally, if zero is true, then set grid[r][c] back to 0.
9. Test your program and correct any errors. It should produce the following output (tab width may vary):  
     
   49 50 51 62 63 68 69 70 71  
   48 53 52 61 64 67 74 73 72  
   47 54 59 60 65 66 75 76 77  
   46 55 58 27 26 25 24 79 78  
   45 56 57 28 5 4 23 80 81  
   44 31 30 29 6 3 22 21 20  
   43 32 33 34 7 2 1 18 19  
   42 39 38 35 8 11 12 17 16  
   41 40 37 36 9 10 13 14 15

†*It was noted by Liberty student David Poe that the Numbrix solving method proposed by the author of the lab will not work when the initial state of the (un-solved) puzzle contains a "1", as the lab suggests you start with a "1" and recursively attempt all possible solutions from 1 to n, pruning non-viable solutions.*

*To make the Numbrix solver work for any starting puzzle (e.g. when solving puzzles found on the internet that may initially contain a "1"), David suggests you start "by calculating the smallest number not already in the puzzle and pruning [starting] with it."*

**(Advanced) Tougher recursion problems**

1. Complete the recursive method public boolean makeBricks(int small, int big, int goal) to solve the famous "Make bricks" problem, seen below:

*We want to make a row of bricks that is****goal****inches long. We have a number of small bricks (1 inch each) and big bricks (5 inches each). Return true if it is possible to make the goal by choosing from the given bricks. This is a little harder than it looks and can be done* ***without any loops****.*

makeBricks(3, 1, 8) >>> true  
makeBricks(3, 1, 9) >>> false  
makeBricks(3, 2, 10) >>> true

1. Write a recursive method (without loops of any kind) that will sum all the elements in a LinkedList of Integers.
2. Complete the method public static boolean sumGroup(int start, int[] n, int target) that determines if it's possible to choose a group of some of the numbers, such that the group sums to the given target.

sumGroup(0, {1, 2, 9}, 10) >>> true

1. A student can do the following activities:

* Do homework in 2 days
* Write a poem in 2 days
* Go on a trip for 2 days
* Study for exams for 1 day
* Play video games for 1 day

A "schedule" of n days can be completed with any combination of the activities above. For example, 3 possible schedules for 7 days are:

homework, poem, homework, play

poem, study, play, homework, study

trip, trip, trip, study

**Write a recursive method** schedule(int n) **that prints all possible schedules for** n **days.**

1. Complete the following method: public int coinDeterminer(int num) that will take the input, which will be an integer ranging from 1 to 250, and return an integer that will specify the **least** number of coins, that (when added) equal *num* (in other words, you're making change).

*Coins are based on a system as follows: there are coins with values 1, 5, 7, 9, and 11. If*num*is 16, then the output should be 2, because you can achieve the number 16 with the coins 9 and 7. If*num*is 25, then the output should be 3, because you can achieve 25 with either 11, 9, and 5 coins or with 9, 9, and 7 coins.*

**(Over 9000) Take this <recursion> - it's dangerous to go alone**

1. Write a method called printSquares() that uses recursive backtracking to find all ways to express an integer as a sum of squares of unique positive integers. For example, the call of printSquares(200) should produce the following output:

1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 8^2 + 9^2

1^2 + 2^2 + 3^2 + 4^2 + 7^2 + 11^2

1^2 + 2^2 + 5^2 + 7^2 + 11^2

1^2 + 3^2 + 4^2 + 5^2 + 6^2 + 7^2 + 8^2

1^2 + 3^2 + 4^2 + 5^2 + 7^2 + 10^2

2^2 + 4^2 + 6^2 + 12^2

2^2 + 14^2

3^2 + 5^2 + 6^2 + 7^2 + 9^2

6^2 + 8^2 + 10^2

Some numbers (such as 128 or 0) cannot be represented as a sum of squares, in which case your method should produce no output. Keep in mind that the sum has to be formed with unique integers (otherwise you could always find a solution by repeatedly adding 1^2).

1. You are given the dimensions of a set of objects. Each object is either a cube, a sphere, or a cylinder. If one object can completely "enclose" another, then the second object can be placed inside the first - the total volume of all objects is thus reduced. Your program should find the placement of objects that gives you the minimum volume, and display this volume.

You cannot put more than one object into any given object. You cannot have an object containing an object containing an object (this simplifies the program).

A cube is specified by its edge length 'a'.   
 Volume of a cube = a3  
A sphere is specified by its radius 'r'.   
 Volume of a sphere = (4 / 3) \* PI \* r3  
A cylinder is specified by its radius 'r' and height 'h'.  
 Volume of a cylinder = PI \* r2 \* h

For example, if three cubes of sides 6, 8 and 12 units are given, the minimum total volume required to hold all the three cubes is 123 + 63.